Financial Data Analysis with Apache Spark

SIMONE NICOSANTI, 0334319, University of Rome Tor Vergata, Italy

This document presents strategies, system architecture and empirical results of financial data analysis with Apache Spark.

1 INTRODUCTION

The aim of the project is to do batch processing on a financial dataset obtained by Fintech Infront Financial Technology, using Apache Spark as data processing framework. This dataset is about trades on three main european markets during the week from november 8th 2021 to november 14th 2021. We considered a reduced version of this dataset: this version consists of 4 million events (compared with 289 million of original dataset) considering 500 equities and indices on european markets of Paris (FR), Amsterdam (NL) e Frankfurt/Xetra (ETR). Events are registered as they have been received and there may be events without payload.

2 SYSTEM ARCHITECTURE

As for system architecture we decided to emulate it with Docker Compose on a single node. We have four main part of the system:

- Client. The Client has been written in Python using PySpark library. An other library used in the client is Redis client library for python, in order to write results on Redis after processing.
- Apache Hadoop. Apache Hadoop has been used as DFS for dataset retrieve and result storage in a distributed way. As for dataset upload to HDFS, it is downloaded from web (dataset) using *curl* command when Namenode container is started, saved in a local file and then uploaded to HDFS. As for results retrieve and storage, they have been done using Apache Spark. We decided to use only one Hadoop datanode, for the purpose of giving more resources to Spark workers.
- Apache Spark. Apache Spark has been used to both pre processing and processing. With regard to pre processing, we used DataFrame API to clean up the dataset from raw text header and not valid lines, while as for processing, we used both RDD and Spark SQL in order to study differences in execution time. Before result saving, RDD have been converted to DataFrame in order to sort them and save in more readable way
- Redis. Redis has been used as key-value storage to store query results. We decided to store only results obtained using RDD API to reduce storage times. All results have been saved using a JSON as value, with the aim of retrieving them all in one with single Redis GET, and query name as key.

After data processing, results have been also saved on a client local directory */Results* which is a Docker Volume attached to client container with the purpose of having results on the host machine, too.

3 PRE PROCESSING

With regard to pre processing we did it with Spark. Unfortunately, due to original dataset format, we could not directrly read it as CSV file because there was a raw text header which was misinterpreted by Spark. As a consequence we decided to:

- Read as TextFile, obtaining a DataFrame of a single column where each row was a CSV line

Author's address: Simone Nicosanti, 0334319, snicosanti@gmail.com, simone.nicosanti.01@students.uniroma2.eu, University of Rome Tor Vergata, Rome, Italy.

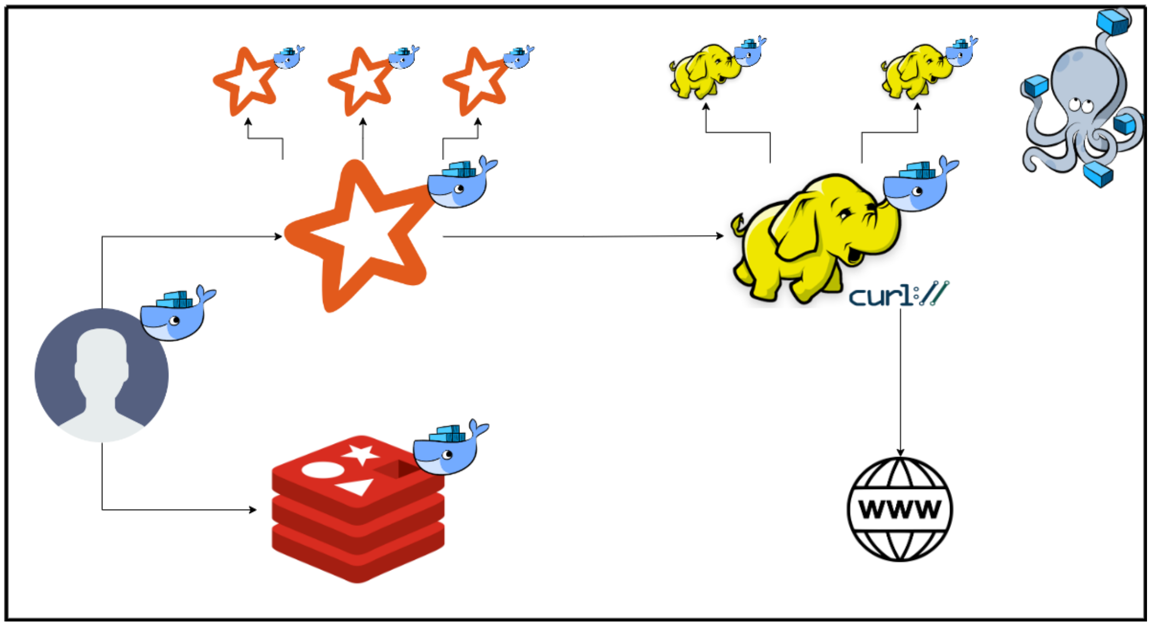


Fig. 1. System Architecture

- Use *split* over the DataFrame splitting by ",", obtaining a DataFrame with all original columns
- Select only interesting columns of CSV

Considering that all queries referenced at most five fields, which are *TradingDate*, *TradingTime*, *ID*, *SecType* and *Last*, we decided to remove all other fields in order to not burden Spark workers. Considering that all queries were based on date and time, we decided to remove all rows which had these fields invalid. After several dataset analysis we noticed that:

- All rows with time equals to *00:00:00.0000* had Last field equals to 0. We decided to consider these rows as reset rows for the price and we removed them.
- There were rows which, for the same Date, Time, ID and SecType had different values for the Last field: we decided to resume these rows in a single row with Last as the average of Last fields.

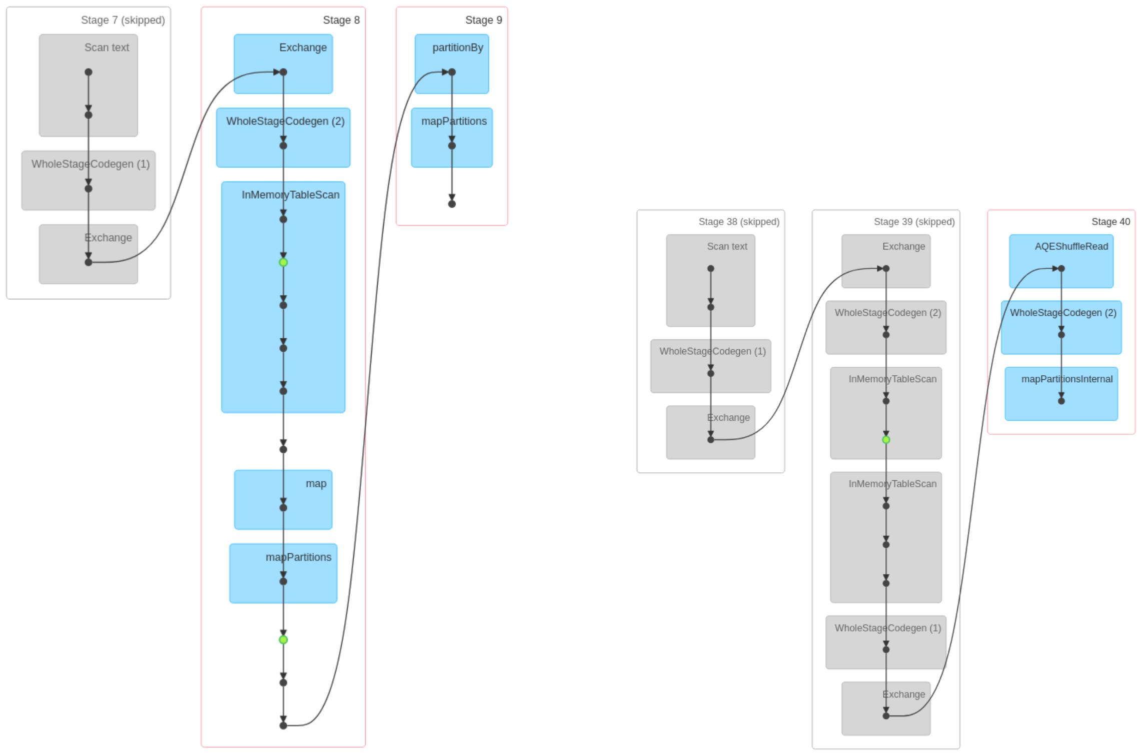
Considering that first and second queries were based based on time hour, we decided to add to DataFrame a redundant column with event hour in order not to compute it again during query processing.

After pre processing operation, we converted DataFrame to RDD and persisted both in order to not compute them again for queries executions.

The final schema for both RDD and DataFrame is as follows:

TradingDate	TradingTime	TradingHour	ID	SecType	Last

Table 1. Final Schema



(a) With RDD (b) With DataFrame

Fig. 2. DAGs for Query 1

4 QUERY 1

Query For each hour, calculate the minimum, average and maximum value of the selling price (Last field) for the only stocks (SecType field with value equal to E) traded on the Paris (FR) markets. In the output also indicate the total number of events used to calculate the statistics. Attention should be paid to events with no payload.

4.1 Query 1 with RDD

As for this query using RDD, we did the following steps:

(1) Filter by ID and SecType
(2) Map to a Pair RDD with key *(Date, Hour, ID)*
(3) AggregateByKey using Spark *StatCounter* which returns statistics of values by key obtaining an RDD like *((Date, Hour, ID) (Stats))*
(4) Map to extract statistics of interest from *StatCounter* result

4.2 Query 1 with Spark SQL

As for this query using SQL we did a simple select filtering by *ID* and *SecType*, grouping by *TradingDate*, *TradingHour* and *ID* and then using aggregate operators *min*, *avg*, *max* and *count*.

5 QUERY 2

Query For each day, for each stock traded on any market, calculate the mean value and standard deviation of their sales price change calculated over a one-hour time window. After calculating the statistical indices on a daily basis, determine the ranking of the best 5 stocks that recorded the best price change during the day and the 5 worst stocks that recorded the worst price change. In the output also indicate the number total number of events used to calculate the statistics.

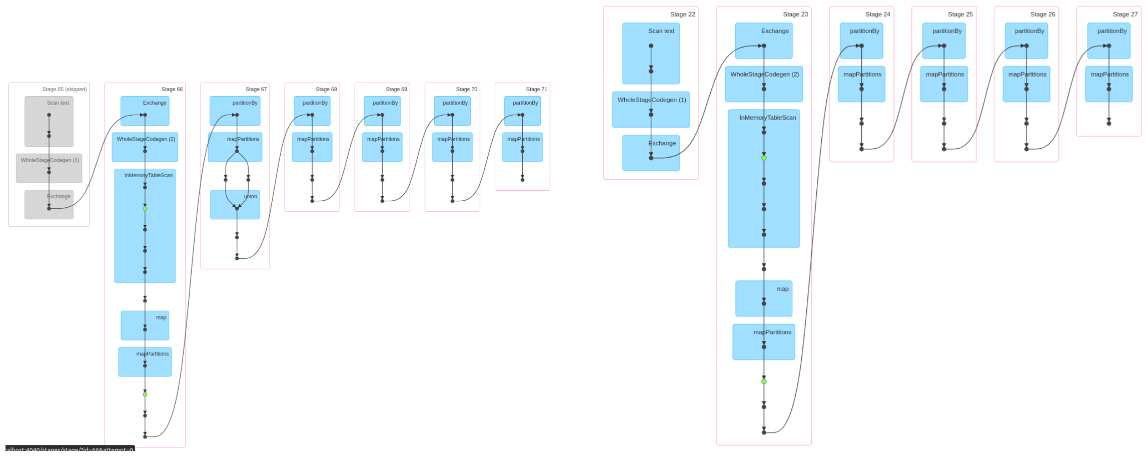
The hardest part of this query was identify the price of each stock at every hour. After a dataset analysis we found out that there was no stock which had a row at time *hh:00:00.0000* so we decided to consider as the price at time *hh:00* the most recent price of a time before *hh:00*. In addition we did not consider variation equals to 0 for those hour windows when there were no trades in order not to have too many 0 values which would have flattened all statistics: for example if there is a trade at 11:58:43.000 with price of 12.43 and then no other trade until 16:45:32:00, we consider 12.43 as the price of 16:00, in order not to have change equals to zero for all hour windows between 12:00 and 15:00.

5.1 Query 2 with RDD

We decied to implement two versions of this Query using RDD, one with a *join* between RDD, the other using a map and a for-cicle.

As for the first variant we have the following steps:

(1) Map to a key-value RDD of type *((Date, ID, Hour) (TradingTime, Last, TradingTime, Last))*.
(2) ReduceByKey in order to find first and last price of the stocks in that time window.
(3) Map to a key-value RDD of type *((Date, ID) (Hour, Last))* obtaining a for each hour window the first and last Trade info.
(4) Map to a key-value RDD to obtain *partialRDD* which gives the first price for each hour window: this has been achieved summing 1 to *TradingHour*
(5) Join *partialRDD* within itself in order to find for a *(Date, ID)* couple all possible couples of *(Time, Last)*: *((Date, ID), ((Hour1, Last1) (Hour2, Last2)))*
(6) Filter for those couples having *Hour1 < Hour2* as we are looking for the greatest time before *Hour2*
(7) Map to get an RDD *(Date, ID, Hour2) (Hour1, Last1, Last2)*
(8) ReduceByKey with key *(Date, ID, Hour2)* in order to find the bigger *Hour1* before *Hour2* and its relative *Last*: after this we achieved an RDD of type *((Date, ID, Hour2), (maxHourBefore, lastBefore, last2))*
(9) Map to find price variation for each time window obtaining an RDD of *(Date, ID), variation*
(10) AggregateByKey in order to find the Variations statistics with Spark StatCounter
(11) Map in order to extract only interesting statistics from StatCounter result. As we need to find the number of tuples used to compute the statistics we cannot directly use the result of *Count* of StatCounter because it counts the number of values used; nevertheless we can use this number and sum 1 to it: in fact, the number of variations used differs from the number of original tuples only by the first tuple of the list of variations. We now have an RDD of *(Date, ID), (mean, stdDev, count)*
(12) Map to an RDD like *(Date, (mean, stdDev, count, ID))* as we need to rank by date



(a) First Variant (b) Second Variant

Fig. 3. DAGs for Query 2 with RDD

(13) GroupByKey to obtain an RDD like *Date, listOf((mean, stdDev, count, ID))*
(14) Maps in order to sort the list for each date and get top 5 and bottom 5. In this way we have a list whose length is of 500 elements at most (which is the max number of equities considered by the dataset); this list is sorted and then we extract some elements. Considering extraction cost as constant we have to consider sorting cost which is $O(n * \log(n))$.
An other way to find the rank would have been looking for max and remove it for five times and similarly for the minimum: this would have led to a cost of $O(10 * n)$ which is grater compared to $O(n * \log(n))$ for $n = 500$
(15) FlatMap the result in order to obtain again a key-value RDD of type *((Date, ID) (Avg, StdDev, Count))*

With regard to second variant we have the following:

(1) First three steps are the same as before
(2) GroupByKey in order to find an RDD of *(Date, ID), listOf((Time, Last))*
(3) Map using a custom function which iterates over the sorted value of each key and computes price variations. In this version we need to sort a list and then iterate over it: the cost of this is not too high because this list contains 24 elements at most.
(4) FlatMap to obtain an RDD like *((Date, ID) , variation)*
(5) Remaining steps are the same as before from step 8 onwards

Thus the two variants differ only in the way they compute the last price before of an hour.

5.2 Query 2 with Spark SQL

As for this query using DataFrame we did the following steps:

(1) Find the last trade info for each hour *hh:00* and then, summing 1 to the hour, find the first price for *hh:00*
(2) Find price couples looking for, for each hour *hh:00*, to the most recent previous trade
(3) Compute for each hour the price change, obtaining a schema of *(Date, ID, Variation)*
(4) Compute statistics grouping by *Date* and *ID*

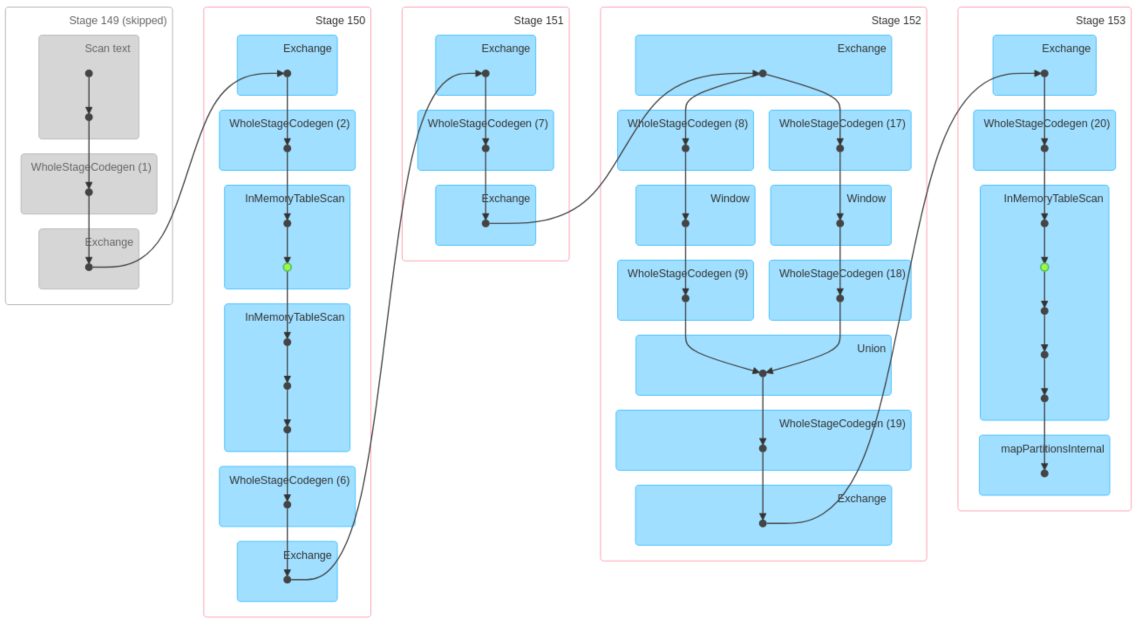


Fig. 4. DAGs for Query 2 with DataFrame

(5) Look for best and worst stocks for each date using nested queries with windows and *row_number()*
(6) Union between best and worst stocks data

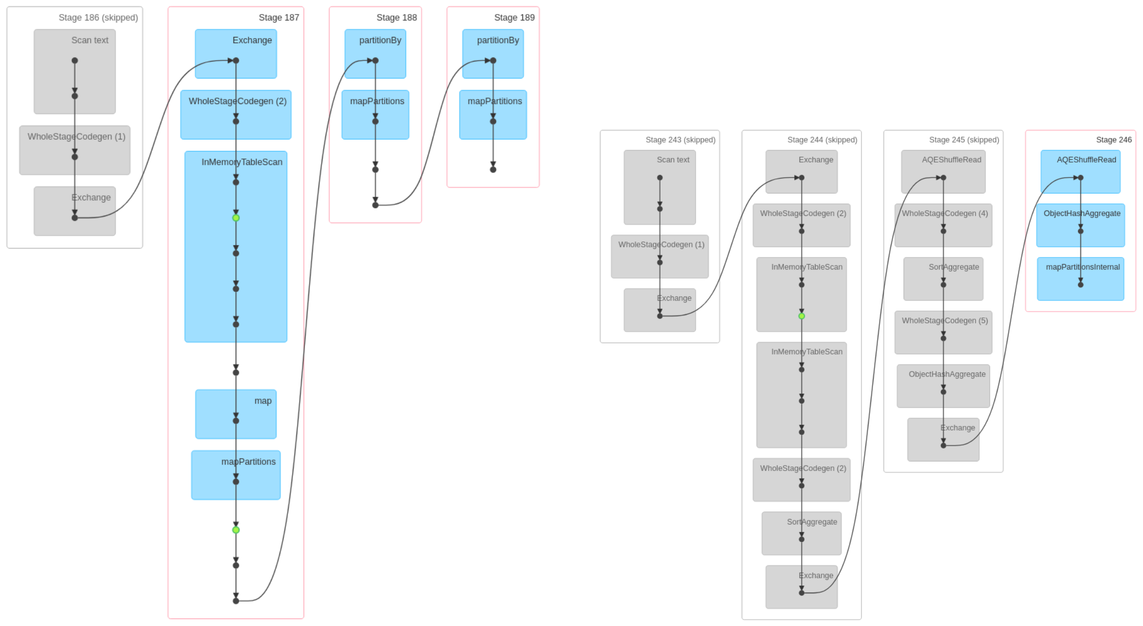
6 QUERY 3

Query For each day, calculate the 25th, 50th, 75th percentile of the change in the price of sale of stocks traded on individual markets. The statistic should be calculated by considering all and only the stocks belonging to each specific market: Paris (FR), Amsterdam (NL) and Frankfurt/Xetra (ETR). In the output also indicate the total number of events used to calculate the statistics.

6.1 Query 3 with RDD

With regard to this query using RDD we did the following steps:

(1) Map to a key-value RDD like *((Date, ID), (Time, Last, Time, Last))*
(2) ReduceByKey in order to find, for each couple *(Date, ID)* the first and last price of the day obtaining an RDD like *(Date, ID) (firstTime, firstPrice, lastTime, lastPrice)*
(3) Filter for those elements with *firstTime != lastTime* in order not to consider those stocks with only one trade during the day
(4) Map to find variation doing the difference between found prices
(5) Map to extract from every *ID* the market of belonging obtaining an RDD with key *(Date, Market)* obtaining an RDD like *(Date, Market) (Variation)*
(6) GroupByKey to find for each *(Date, Market)* the list of variations



(a) With RDD (b) With DataFrame

Fig. 5. DAG for Query 3

(7) Map to sort this list and then find percentiles.
We computed percentiles using a custom function named *computePercentile* which takas as input a list and a float for the percentile value. This function multiplies list length with percentile value and then if the product is an integer, it computes the percentile as the average of two adjacent elements of the list, otherwise it returns the element whose index is the rounded product value.

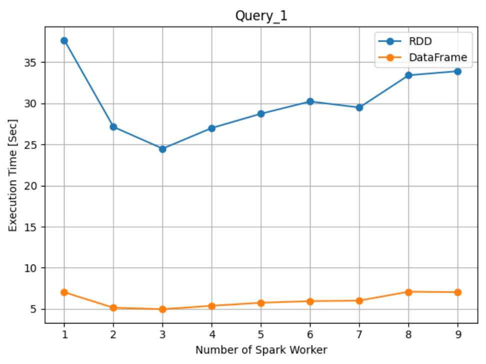
6.2 Query 3 with Spark SQL

As for this query using DataFrame we followed the same logic as the RDD version:

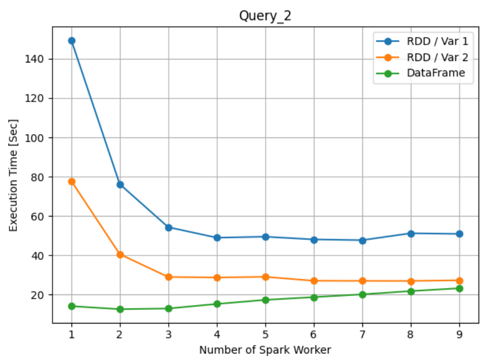
(1) Find, using a nested query, for each couple *Date, ID*, the first and last trade time excluding those with same min and max time as they had no variation during that date.
(2) Find using a double *Join* the first and last price of each Stock
(3) Find price change over the day
(4) Extract Market from *ID*
(5) Find percentiles grouping by *Date* and *Market* and then aggregating with *approx_percentile*

7 EMPIRICAL RESULTS

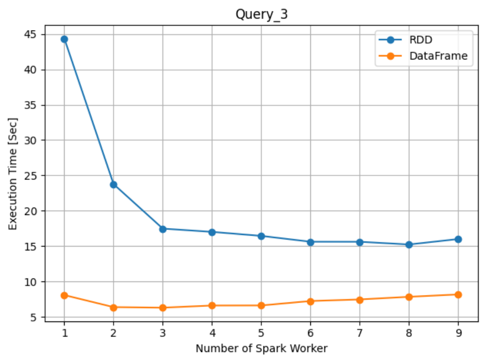
All execution times have been taken without considering either pre process times to convert the read file and post processing time to prepare results for saving on local file system, Redis and HDFS. Times have been taken using Python



(a)



(b)



(c)

Fig. 6. Execution Times Charts

time library, taking time before and after a *collect()* on the result. All experiments have been done using Docker Compose on a single node.

We executed five runs for each query; after the execution of the three queries both with RDD and SQL we closed and reinitialized *SparkContext* and *SparkSession* with the aim of deleting previous execution context. We considered Spark Workers configured as follows:

- SPARK_WORKER_MEMORY=1G
- SPARK_WORKER_CORES=1

As we can see from the charts, a query executed using SQL is always faster than the same query executed using RDD, thanks to Spark SQL Catalyst optimizer.

With regard to RDD processing, we can see how increasing worker nodes produces a significant reduction of execution time until a certain number of workers is reached: from that number on we have a plateau for Query 2 and Query 3, while we have a time increment for Query 1. The initial decrease is because, increasing the workers number, we increase the parallelism degree and, as a consequence, we have a faster processing. As for Query 1 we can say that after a certain number of workers, shuffling time overrides the execution time. As for *Query 2*, both versions of the query using RDD have the same trend: in fact they differ mainly in the *Join* operation and subsequent processing of tuples until obtaining variations, but from that moment on they do the same processing. This is both because *Join* is very expansive operation and because *Join* produces an high number of tuples that are subsequently processed.

With regard to queries done with DataFrame, we have a first decrease when we increment workers number from one to two and then we have a more or less constant trend or a slow growth.

Finally we can say that these charts show only a partial vision of time trend to the increment of workers: indeed, due to the fact that the execution is on a single node, the network latency is near zero and it is not considered in these charts or it is poorly represented.

DataStructure	**Variant**	**WorkersNum**	**Avgs**
RDD	1	1	149.50933299064636
RDD	1	2	76.23494944572448
RDD	1	3	54.33999037742615
RDD	1	4	49.00660648345947
RDD	1	5	49.49895539283752
RDD	1	6	48.114559936523435
RDD	1	7	47.737472200393675
RDD	1	8	51.23345718383789
RDD	1	9	50.91918497085571
RDD	2	1	77.8666223526001
RDD	2	2	40.61496090888977
RDD	2	3	29.009704542160033
RDD	2	4	28.772153091430663
RDD	2	5	29.084913349151613
RDD	2	6	27.09864511489868
RDD	2	7	27.03141584396362
RDD	2	8	27.003732109069823
RDD	2	9	27.328655624389647
DataFrame	1	1	14.175981426239014
DataFrame	1	2	12.640813207626342
DataFrame	1	3	12.982159852981567
DataFrame	1	4	15.297656393051147
DataFrame	1	5	17.363919734954834
DataFrame	1	6	18.76537141799927
DataFrame	1	7	20.168613052368165
DataFrame	1	8	21.849239540100097
DataFrame	1	9	23.2651584148407

Table 2. Query 2 Averages

DataStructure	Variant	WorkersNum	Avgs
RDD	1	1	37.66635408401489
RDD	1	2	27.113973569869994
RDD	1	3	24.475249528884888
RDD	1	4	26.980969047546388
RDD	1	5	28.71597752571106
RDD	1	6	30.200550365448
RDD	1	7	29.468757247924806
RDD	1	8	33.383986949920654
RDD	1	9	33.881511497497556
DataFrame	1	1	7.032466316223145
DataFrame	1	2	5.156687450408936
DataFrame	1	3	4.975139999389649
DataFrame	1	4	5.387648010253907
DataFrame	1	5	5.756169605255127
DataFrame	1	6	5.954088020324707
DataFrame	1	7	6.016712236404419
DataFrame	1	8	7.096331453323364
DataFrame	1	9	7.039819765090942

Table 3. Query 1 Averages

DataStructure	Variant	WorkersNum	Avgs
RDD	1	1	44.362346696853635
RDD	1	2	23.758493089675902
RDD	1	3	17.46725115776062
RDD	1	4	17.00481786727905
RDD	1	5	16.435511207580568
RDD	1	6	15.611356401443482
RDD	1	7	15.604358530044555
RDD	1	8	15.22194037437439
RDD	1	9	15.991070985794067
DataFrame	1	1	8.062828922271729
DataFrame	1	2	6.3605152606964115
DataFrame	1	3	6.280957126617432
DataFrame	1	4	6.587774848937988
DataFrame	1	5	6.597464084625244
DataFrame	1	6	7.22327184677124
DataFrame	1	7	7.447644472122192
DataFrame	1	8	7.8194786548614506
DataFrame	1	9	8.153690719604493

Table 4. Query 3 Averages